

More About Functional Programming in Scala



Recursion

- Functional programming paradigm prefers recursion to iteration
 - Less dependency on mutable state
 - Supports declarative style
- Many algorithms can be expressed in a recursive way

```
scala> def factorial ( i: Int ): BigInt = i match {  
  |   case 0 => 1  
  |   case _ => i * factorial(i - 1)  
  | }  
factorial: (i: Int)BigInt  
  
scala> factorial(0)  
res163: BigInt = 1  
  
scala> factorial(10)  
res168: BigInt = 3628800  
  
scala> factorial(10000)  
java.lang.StackOverflowError  
...
```

Tail Recursion and Tail Call Elimination

- Every recursive call needs a new stack frame
 - Deep recursion can cause stack overflow error
- Tail recursion occurs when
 - Recursive call is last action of a function
 - Result of the function call is returned directly

```
scala> def gcd ( a: Int, b: Int ): Int = if ( b == 0 ) a else gcd( b, a % b )  
gcd: (a: Int, b: Int)Int
```

```
scala> gcd(14,35)  
res170: Int = 7
```

- Tail call elimination is an optimisation that can be used here
 - Reuse current stack frame for recursive call
 - Recursion effectively becomes iteration
-

Tail Recursion and Tail Call Elimination


- Not all algorithms are tail recursive by default
 - Scala compiler will generate tail call elimination code when possible
 - Otherwise normal recursion
- `@tailrec` annotation causes compilation failure if no tail recursion in annotated function

```
scala> @tailrec
| def factorial( i: Int ): BigInt = i match {
|   case 0 => 1
|   case _ => i * factorial(i-1)
| }
<console>:16: error: could not optimize @tailrec annotated method factorial:
           it contains a recursive call not in tail position
           case _ => i * factorial(i-1)
                       ^
```

Tail Recursion and Tail Call Elimination

- Non tail-recursive function can often be transformed into tail-recursive
 - Introduce argument to pass accumulated value to next call

```
scala> def factorial( n: Int ): BigInt = {  
  |   @tailrec def fact( n: Int, accum: BigInt ): BigInt = n match {  
  |     case 0 => accum  
  |     case _ => fact ( n-1, n * accum )  
  |   }  
  |   fact(n,1)  
  | }  
factorial: (n: Int)BigInt  
  
scala> factorial(10000)  
res176: BigInt = 284625968091705451890641321211986889014805140170279923079417999427  
4411340003764443772990786757784775815884062142317528830042339940153518739052421  
...
```



Use nested method
to avoid changing
external interface

Currying

- Currying is a technique to transform a function taking 2 or more arguments into a composition of functions taking 1 argument

Function that takes one
Int parameter and returns
another function from Int to Int

Function that adds 2 to its
Int argument

```
scala> val add = ( a: Int, b: Int ) => a + b  
add: (Int, Int) => Int = <function2>
```

```
scala> add.curried  
res182: Int => (Int => Int) = <function1>
```

```
scala> add.curried(2)  
res183: Int => Int = <function1>
```

```
scala> add.curried(2)(3)  
res184: Int = 5
```

Partial Function Application

- Currying a function allows partial application
 - Generate a function that effectively fixes the first argument(s)
 - Use this generated function further

```
scala> val add2 = add.curried(2)
add2: Int => Int = <function1>
```

```
scala> add2(3)
res185: Int = 5
```

```
scala> val foo = (a: Int, b: Int, c: Int) => a + b + c
foo: (Int, Int, Int) => Int = <function3>
```

```
scala> def add23 = foo.curried(2)(3)
add23: Int => Int
```

```
scala> add23(4)
res186: Int = 9
```


Lazy Parameter Evaluation

- AKA Pass By Name
- Suppresses evaluation of actual parameter expression until parameter is used in the function

```
def showTheTime ( t: java.util.Date ) = {  
  println(s"The time is $t")  
  Thread.sleep(1000)  
  println(s"The time is $t")  
}
```

```
scala> showTheTime( new java.util.Date )  
The time is Sat Aug 13 20:52:49 CST 2016  
The time is Sat Aug 13 20:52:49 CST 2016
```

Evaluated on call
Value used inside
function




Lazy Parameter Evaluation

- Parameter passing is normally by value
- Actual parameter expression is evaluated as part of call

```
def showTheTime ( t: => java.util.Date ) = {  
  println(s"The time is $t")  
  Thread.sleep(1000)  
  println(s"The time is $t")  
}
```

```
scala> showTheTime(new java.util.Date)  
The time is Sat Aug 13 21:23:56 CST 2016  
The time is Sat Aug 13 21:23:57 CST 2016
```

Evaluated each time
parameter is used
during the function call



Managing Resources

- Curried function/method definition and pass by name parameters allow definition of powerful structures within a program
 - Resource management

```
def sync ( l: Lock ) ( f: => Unit ) = {  
  l.acquire()  
  try {  
    f  
  } finally {  
    l.release()  
  }  
}
```

Code in f is
evaluated here

Must be passed by name

```
val myLock = new Lock()  
sync ( myLock ) {  
  println("I am holding the lock")  
}
```

Streams – Lazy Collections

- Streams present a similar abstraction to Lists
 - Streams are *lazy*
 - Elements are evaluated/calculated only when required
 - List requires all elements to be present
 - Useful for working with sequences whose elements are not (yet) available
 - Also infinite sequences
 - E.g. prime numbers
-

Defining Streams

- Streams are defined in a similar way to Lists

- A Stream is

- Empty
 - An element followed by another Stream

- But...

- The tail is calculated only when required

```
object Stream {  
  def cons[T](hd: T, tl: => Stream[T]) =  
    new Stream[T] {  
      def isEmpty = false  
      def head = hd  
      lazy val tail = tl  
    }  
  
  val empty = new Stream[Nothing] {  
    def isEmpty = true  
    def head = throw  
      new NoSuchElementException("empty.head")  
    def tail = throw  
      new NoSuchElementException("empty.tail")  
  }  
}
```

Defining Streams

- Defining a Stream of Int

```
scala> val str1: Stream[Int] = Stream.cons(0, Stream.cons(1, Stream.empty))  
str1: Stream[Int] = Stream(0, ?)
```

- Use operator #:: as alternative notation

```
scala> val str2 = 0 #:: 1 #:: empty  
str2: scala.collection.immutable.Stream[Int] = Stream(0, ?)
```

- ? in toString output indicates tail is calculated on demand
-

Defining Streams

- Seq[A] can be converted to a Stream [A]

```
scala> val l1 = List(1,2,3,4,5,6,7,8)
l1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8)

scala> l1 toStream
res171: scala.collection.immutable.Stream[Int] = Stream(1, ?)

scala> 1 to 10 toStream
res170: scala.collection.immutable.Stream[Int] = Stream(1, ?)
```

- Elements will be pulled through stream as required
-

Defining Streams

- Stream can be defined explicitly
 - Function to calculate "next" element

```
scala> def rangeToStream ( from: Int, to: Int ): Stream[Int] = {  
  |   if ( from >= to ) Stream.empty  
  |   else Stream.cons(from, rangeToStream(from+1, to) )  
  | }  
rangeToStream: (from: Int, to: Int)Stream[Int]
```

```
scala> rangeToStream(1,4)  
res164: Stream[Int] = Stream(1, ?)
```

```
scala> rangeToStream(1,4) take(2) toList  
res165: List[Int] = List(1, 2)
```

Infinite Streams

- Streams do not require a finite length

```
scala> def streamOfSquares(from: Int): Stream[Int] =  
      (from*from) #:: streamOfSquares(from + 1)  
streamOfSquares: (from: Int)Stream[Int]
```

- or

```
scala> def streamOfSquares( from: Int ): Stream[Int] =  
      Stream.cons( (from*from), streamOfSquares(from+1))  
streamOfSquares: (from: Int)Stream[Int]
```

Infinite Streams


- Operations on infinite stream return a stream

```
scala> streamOfSquares(1) map ( _ + 1 )  
res177: scala.collection.immutable.Stream[Int] = Stream(2, ?)
```

- Use other methods to "materialize" the stream

```
scala> val ss = streamOfSquares(1)  
ss: Stream[Int] = Stream(1, ?)  
  
scala> ss take(3) toList  
res5: List[Int] = List(1, 4, 9)  
  
scala> ss  
res6: Stream[Int] = Stream(1, 4, 9, ?)
```

Notice Stream elements
are now memoized



Useful Stream Methods

- `Stream.empty`
 - Return an empty stream
 - `Stream.from(n)`
 - Return a stream that supplies `Int` values starting at `n` and incrementing by 1 each time
 - `Stream.continually`
 - Return a stream that uses the supplied expression to calculate the next element
 - `Stream.range`
 - Return a stream that supplies the values defined in the range
 - Like the `rangeToStream` method shown earlier
-

Partial Functions

- Partially defined, not partially evaluated (curried)
 - Function not defined over its entire domain
 - E.g sqrt over Int
 - Scala provides type level support for partial functions
 - PartialFunction type is subtype of Function1
 - Defines two additional methods
 - `def isDefinedAt(a: A): Boolean`
 - `def orElse(that: PartialFunction[A,B]):
PartialFunction[A,B]`
-

Defining a Partial Function

- Case alternatives

```
scala> val pf: PartialFunction[Int, String] = {  
  |   case 1 => "one"  
  |   case 2 => "two"  
  |   case 3 => "three"  
  | }  
pf: PartialFunction[Int,String] = <function1>  
  
scala> pf.isDefinedAt 2  
res55: Boolean = true  
  
scala> pf(2)  
res56: String = two  
  
scala> pf.isDefinedAt 4  
res57: Boolean = false  
  
scala> pf(4)  
scala.MatchError: 4 (of class java.lang.Integer)  
  at  
scala.PartialFunction$$anon$1.apply(PartialFunction.scala:248)  
  ...
```

Defining a Partial Function

- Implement the required methods

```
scala> object SquareRoot extends PartialFunction[Int, Double] {  
  |   def apply ( i: Int ) = if ( i >= 0 ) Math.sqrt(i) else  
  |       throw new IllegalArgumentException("...")  
  |   def isDefinedAt(i: Int ) = ( i >= 0 )  
  | }  
defined module SquareRoot  
  
scala> SquareRoot isDefinedAt(4)  
res62: Boolean = true  
  
scala> SquareRoot(4)  
res63: Double = 2.0  
  
scala> SquareRoot isDefinedAt(-1)  
res64: Boolean = false  
  
scala> SquareRoot(-1)  
java.lang.IllegalArgumentException: Negative sqrt not cool  
    at SquareRoot$.apply$mcDI$sp(<console>:17)
```

Map as a Partial Function

- Map type is a mapping from keys to values
 - Not every value from the key type may have a mapping
 - Hence Map is a partial function

```
scala> val squares = Map( 1->1, 2->4, 3->9)
squares: scala.collection.immutable.Map[Int,Int] = Map(1 -> 1, 2 -> 4, 3 -> 9)

scala> squares.isDefinedAt(2)
res190: Boolean = true

scala> squares.isDefinedAt(4)
res191: Boolean = false

scala> squares(2)
res192: Int = 4

scala> squares(4)
java.util.NoSuchElementException: key not found: 4
  at scala.collection.MapLike$class.default(MapLike.scala:228)
  ...
```

Composing Partial Functions

- Composition of partial functions means providing mappings for previously undefined values
 - `orElse` method
 - Does not overwrite existing values if key already present

```
scala> val moreSquares = squares orElse Map( 4-> 16, 5->25 )
moreSquares: PartialFunction[Int,Int] = <function1>
```

```
scala> moreSquares.isDefinedAt(4)
res195: Boolean = true
```

```
scala> moreSquares(4)
res196: Int = 16
```
